The Design Space for Data Replication Algorithms in Interactive Groupware*

Sumeer Bhola Mustaque Ahamad
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
{sumeerb, mustaq}@cc.gatech.edu

Technical Report GIT-CC-98-15

Abstract

Interactive (Synchronous) Groupware encompasses a wide range of applications, like collaborative whiteboards, text editors, engineering CAD (Computer Aided Design), Distributed Virtual Environments, and multi-player games. A very critical requirement for all these applications is the need to share data, which can be replicated to provide better responsiveness, fault-tolerance and scalability. Despite the existence of many systems and distributed algorithms for replication in interactive groupware, there is little agreement on the shared data abstraction to provide, or on the general algorithmic approach. Similarly, there is general lack of uniformity in the terminology used to describe such algorithms. This paper attempts to rectify this situation by (1) Describing the fundamental data requirements of interactive groupware, (2) Developing a data abstraction which is appropriate for meeting these requirements, (3) Developing a general algorithmic structure and terminology to express the characteristics of algorithm instances, and (4) Classifying the algorithms proposed in the research literature.

*This work was supported in part by an IBM fellowship and NSF grant CCR-9619371

1 Introduction

Synchronous or Interactive[1] Groupware is a rapidly growing area of commercial and research interest. This includes a wide range of applications, from traditional ones like whiteboards and text editors, to emerging ones like engineering CAD (Computer Aided Design), DIS (Distributed Interactive Simulation), and multi-player games. By our definition, the term Interactive Groupware stands for *any* collection of applications, that allow a potentially distributed group of users to collaborate or compete on a task at the same time. A very critical requirement for all these applications is the need to share data, and that is the focus of this paper.

The interactive nature of such applications requires that the effect of a user's action is seen by himself (response time or RT) as well as other users (user to user time or UUT) in a timely manner. However, the problem of providing appropriate RT and UUT is increasingly difficult as groupware is deployed in a wide-area distributed environment like the Internet, where high communication latencies are common. In particular, end-users are increasingly accustomed to direct manipulation user interfaces, which typically require response times on the order of 50-100*ms*. However, due to the fundamental limitation of the speed of light, the round-trip delay to the far side of the planet is at least 200*ms*. Other causes of higher message latencies are the use of mobile wireless computers, and when connecting from home through modems.

Though UUT by its very nature has to depend on the message latency between a pair of users, it is possible to make RT less dependent on message latency by using replication of the shared data at user sites. This has the potential to reduce response time for actions that read or modify this data, since a user's action can be executed on her local replica. There is general agreement in the research community on the need for replication as can be seen by the large number of groupware systems [9, 12, 17, 16, 18, 20, 5] that support replication. In addition to reducing RT, replication can also provide resilience to failures, which can be common in a distributed environment. Finally, many factors, including smaller size of messages and batching of modifications, can allow replication to consume fewer network resources than a centralized approach.

Data can be replicated at all the sites which have users interested in that data or at some strategically located sites, depending on the memory resources available. Note that replication at all interested users sites does not consume more network resources than replication at only some sites, as notifications about the change in state of the data still need to go to all these sites, to update their views. Therefore, in this paper we will only be concerned with replication at all interested sites, i.e. *full* replication.

One common misconception about replication is that it makes life harder for the application programmer. We illustrate why this is not so by considering the common way in which these applications are structured. The application, running on a certain site, receives a user action expressed as a windowing system event, and translates this

[1] We will use the term Interactive instead of Synchronous or Real-time. Also groupware refers to Interactive Groupware from now on, unless explicitly qualified.

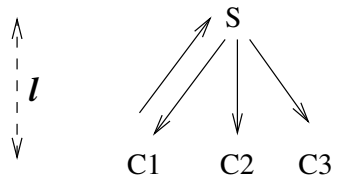


Figure 1: Total Ordering through a Server

into an operation that needs to be executed on the shared data. It then issues this operation to the underlying data management system which executes this operation on the local replica and the other replicas. Additional complexity, if any, for the application programmer, comes from the potential for concurrency. In particular, between the translation step and execution step of this operation, another operation (probably one that is concurrently issued by another site) can be executed on the shared data. And this can happen even in single copy systems ! For example, consider the text editing example used in [9]. The text buffer is represented as an array of characters, and 2 concurrent operations are issued to insert x characters at positions 5 and 10 respectively. if the first operation gets to execute earlier, the second insert should happen at position $10 + x$. Operation transformations were proposed by Ellis and Gibbs in [9] (and formalized by Cormack in [7]), to solve this problem of preserving the intent of the user, and also to guarantee replica convergence. We will discuss both aspects later in the paper (Section 7.1), and assume for now that no operation transformations are needed.

1.1 The Simplest Replication Approach

With replication comes the associated problem of maintaining the replicas consistent with respect to each other. The simplest approach is to ensure that all operations that modify the data get executed in the same order at all replicas, by ordering them through a server (as depicted in figure 1). Read-only operations can still be executed instantaneously at the local replica. Despite the simplicity of this scheme, it suffers from some major drawbacks. Firstly, assuming that the one-way communication latency between every pair of sites is l, the RT and UUT is $2l$. This may be too high in many situations where l is large or widely varying. Secondly, the system is not fault-tolerant to server failure, and the server can be a performance bottleneck. Finally, applications cannot batch operations before sending to the server, as it would affect RT. Batching of operations [5] can be very useful to reduce network traffic when other users are not really interested in each others work.

The algorithms proposed for synchronous groupware all reduce by some degree the dependence on the server, i.e. the algorithms are more distributed. Most of them have a best case RT and UUT that is less than $2l$, but whether they provide an average case lower RT, UUT than the server based approach depends on the distribution of the inter-site network latencies and the application scenario. Though evaluation of such algorithms is not common, and is not the focus of this paper, some initial work is described in [4].

Although many systems and distributed algorithms have been proposed for replication in groupware, there seems to be little agreement on the shared data abstraction to provide, or on the general algorithmic approach. Similarly, there is general lack of uniformity in the terminology used to describe such algorithms. Researchers have approached the problem from various angles, some coming from a database background, some from a general distributed computing background, and some from high performance computing (for example, distributed shared memory). Each of these approaches has missed some critical need of groupware. The first goal of this paper is to illustrate the data sharing requirements and come up with a data abstraction which captures and extends the power of the existing abstractions. Next, we describe a general algorithmic structure for implementing this abstraction, and discuss the important design choices. This is used to roughly classify the algorithms described in the groupware research literature, even though some implement a different abstraction than the one we propose. The goal of this paper is not to claim that all the problems have been solved, or to stifle algorithm research for these applications by restricting them to a certain general structure. On the contrary, we hope that a better understanding of the design space, and a common terminology for describing the working of these algorithms, will stimulate research.

Section 2 illustrates the data characteristics of these applications using two examples. These characteristics are used to motivate the set of objects abstraction, which is described in Section 3. Section 4 discusses why replicated databases cannot be easily adapted to meet our requirements. Section 5 describes the general structure of the algorithms, including the global ordering and timestamping properties, and the issue of replica-view coupling. This is used in section 6 to classify many algorithms proposed in the groupware research literature. Section 7 discusses certain secondary, but important issues, including operation transformation, and granularity of access. Finally, we conclude in section 8.

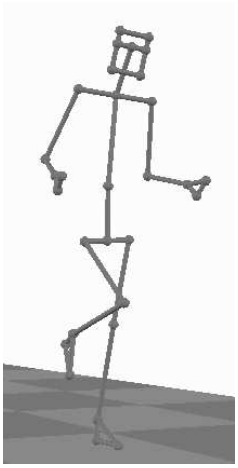
2 Shared Data Characteristics

This section describes the characteristics of the shared data. We first illustrate with some examples of application scenarios.

2.1 Application Scenarios

The first example is of the collaborative design of a key-frame for a computer animation, which can be part of the development of an animated movie, or the development of the visual behavior of a complex entity (e.g. a virtual human) in an interactive single-user/multi-user virtual world. The second example is of such a distributed virtual world in action, specifically of interaction between distributed entities, some of which could be under user control and some autonomous.

These two examples illustrate how multi-user interaction can be important for all stages of a groupware application, from design and development to actual use.



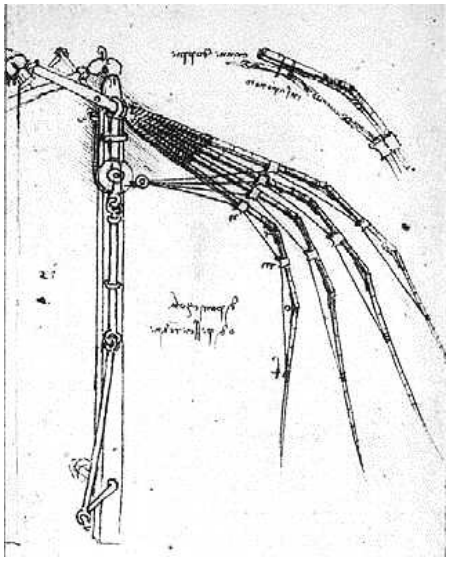


Figure 2: Articulated Models: Stick Model of Human, Leonardo da Vinci's Wing

Key-frame/Snapshot design We assume that the key-frame design involves arrangement of multiple rigid entities in a certain 3-dimensional space. In addition to simple rigid entities like a table or a bottle, there are other more complex entities like humans or synthetic creatures. Such complex entities are represented by an *articulated model*[11]. An articulated model is a collection of objects connected together by joints in a hierarchical, tree-like structure. Figure 2 shows a simple stick figure model of a human, which can be internally represented as an articulated model. Rotation about the elbow joint in this model will affect not only the position of the lower arm but also the position of the objects below it in the hierarchy i.e. the hand and fingers. Each joint has certain degrees of freedom, and for a certain axis of rotation there is a constraint on the relative angle between the two sticks connected by that joint. For example, the elbow joint has 2 degrees of freedom, with a range of rotation from 0 to 180 degrees for both axes.

A key-frame design can involve many tasks such as, designing an articulated model of a complex entity by adding and deleting sticks and joints, adding and deleting entities from the 3D space, translating entities to certain points in space, and orienting entities or parts of entities by rotating about joints.

Distributed Virtual World A distributed virtual world has many entities, some under direct user control, some autonomous with programmed behavior patterns, and some only controlled by the laws of physics[2](e.g. a tennis ball). See chapter 1 of [19] for a good introduction. Most entities have a primary site which controls that entity, for example, for a user controlled entity the primary site is the site of that user. Primary sites for autonomous entities may be chosen so as to balance the load. A site controlling an entity composed of multiple objects (e.g. a complex entity) has to update a subset of these objects in response to user input, or programmatic control (e.g. to simulate a human walking). In addition, non-primary sites can issue updates to an entity in the case of entity interaction, for

[2]Or whatever be the fundamental law of that world.

example, collision between entities. Proper ordering between such concurrent updates is important to determine the future behavior of the interacting entities. For example, the collision of a tennis ball with a racquet must be identically observed at all sites, including the velocity of each entity, and the exact point of impact. Entities may also be created and deleted, for example, firing of a missile causes creation of a new missile entity, and after hitting a target this entity is deleted.

2.2 Characteristics

The following are the data characteristics of groupware.

1. *Dynamic state and size of shared data* : The amount of data being shared is changing during the collaboration, with new data items being added and existing ones being deleted. Also, the state of this data is continuously changing.

2. *Atomicity of access*: Users can atomically read and write a subset of this data. For example, rotating an elbow joint changes the coordinates of the lower arm, hand and fingers. By atomicity, we do not mean failure atomicity as in transactions, but that the execution of an operation is not interleaved with other operations at a replica. Failure atomicity, as implied by databases, is not needed because replication gives us fault tolerance. Also, as the replicas are not persistent, we do not need a heavyweight distributed commit protocol like 2-phase commit.

3. *Data-dependent access patterns*: In some cases, the subset of the shared data that is accessed by an atomic operation[3] depends on the state of the data when the operation is executed. For example, assume that while the arm of a human model is moving, the hand grabs a door. Future updates on the arm will also need to update the position of the door, and the degrees of freedom of the door can constrain the direction in which the arm can move.

   Despite such data-dependent access patterns, in many cases it is possible to quite accurately predict the access information when an operation is issued.

4. *Short, Incremental Operations*: The execution time for each operation is short, as users want to see the results of the actions of other users.

5. *Commutativity*: Many operations may be commutative, and exploiting the commutativity of operations can allow the data management/consistency algorithm to relax the order in which operations are executed at replicas. Operations are commutative when they access different parts of the data, or, even if they access the same data, their accesses commute. For example, in an articulated model of a human, concurrent rotations about the elbow and the wrist are commutative (if there are no external constraints on the motion).

[3] All operations are atomic, so we will not qualify it from now on.

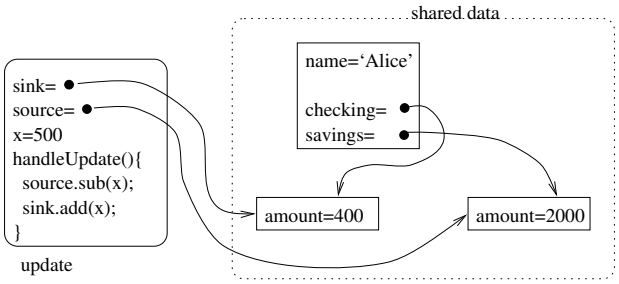


Figure 3: Updating an o-set

3 Set of Objects Abstraction

A *set of objects* (referred to as an *o-set*) abstraction is used to represent a collection of *related* shared data objects. By objects, we do not necessarily mean objects in the OO (Object-Oriented) sense, we mean that they are composite i.e. composed of primitive objects like integer, real etc. These objects can have pointers to other objects in the same set and these pointers can form cycles. Pointers provide a convenient way for the application programmer to express shared data-structures. Sites in a collaboration can dynamically *connect* or *disconnect* from an o-set and objects can be *added* to and *deleted* from the o-set. The sites that are currently connected to the o-set are referred to as *participants* in the o-set. An o-set is fully replicated at all the participant sites. Partial replication of an o-set is outside the scope of this paper. Objects are modified using *updates* that can read and write a subset of objects in the o-set. The main operations on an o-set are:

- `addObject(Object[] o)` Adds one or more objects to the o-set. The objects being added can have pointers to each other.
- `deleteObject(Object[] o)` Delete one or more objects from the o-set. The application is responsible for ensuring that there are no dangling pointers from the remaining objects in the o-set to the subset being deleted.
- `updateSet(Update e)` This issues an update to the objects in the o-set which can read and write a subset of these objects. The update object, e, encapsulates the actual update i.e. it has the state and logic to carry out the update. This state includes pointers to certain objects in the o-set, and the update can only access objects reachable from these pointers. The `handleUpdate` method of e, when executed, performs the actual update. Figure 3 shows an update object with its associated `handleUpdate` method which modifies two of the three shared objects in the o-set.

These three operations are asynchronous, i.e. they are scheduled for execution locally and remotely at some time in the future by the underlying replica management algorithm. The delete operation requires coordination between

processes before the delete is done, to ensure that a process does not receive an update on an object that has already been deleted. In addition to the above three operations, there are read-only operations which are only executed on the local site. These are needed, for example, to refresh the *view* on the data (the GUI showing the data to the user). We limit our discussion to only the add and update operations. Note that an application may be connected to multiple o-sets, but that different o-sets do not share objects.

3.1 Access Information for Updates

As updates may be issued concurrently by multiple sites, the underlying replica management algorithm needs to ensure some ordering of these updates, so that each replica state is consistent. We will discuss consistency in detail in section 5, but at minimum we require that the replicas should converge.

To ensure convergence, the consistency algorithm needs to know what objects are accessed by an update, and how they are accessed. It is useful to allow two options of specifying this information. *Predeclared* access information is provided by an application when it issues an update, and gives a superset of the objects that the update may access. However, for data-dependent operations, the access information can be hard to accurately predict. In this case, predeclared information is not provided and the update has to be executed to determine access. As the update executes, whenever it is going to access an object in some way it hasn't accessed before, it notifies the consistency algorithm by invoking a method. When the notification method returns, the update can continue with its execution. We refer to this as *execution-time* access information.

We will only consider read, write access information for the objects i.e. the application or update informs the consistency algorithm about which objects are read, and which are written. This may seem very restricted, because it seems not to capture commutativity and other semantics which can be used to relax the operation ordering. For example, two concurrent operations that increment the same integer will access that integer in write mode, and therefore will be ordered. However, we will see later that due to the nature of the general algorithmic structure (specifically that operations are not interleaved at a replica, and only one notification is sent per operation), it is possible to specify access information that is factually incorrect, but enough to guarantee convergence. For example, in the above case we can specify that the two increments are just reading the integer. Another operation which just overwrites the integer value with a new value can be specified to be writing the integer, and this is enough to guarantee convergence.

Generalization beyond read, write accesses can be done by defining a *compatibility matrix* which expresses which combination of accesses conflict. Section 2.5 of [3] discusses this issue, and is also proposed for collaborative applications in [14].

3.2 Decoupling Atomicity and Locking

In the o-set abstraction, an operation encapsulates the unit of computation that should be performed atomically. Access information, supplied in two possible ways, is used by the replica management algorithm to provide consistency by imposing a certain ordering on the operations. We will refer to such algorithms as *Ordering* algorithms.

This approach is different from explicit locking by the application to achieve both atomicity and consistency. The explicit locking approach is undesirable because of several reasons.

1. The semantics of the locks (for example, optimistic or pessimistic locks) used have to be fixed when the application is written. This drastically reduces the flexibility in choosing a consistency algorithm at runtime. The following example shows how the receipt of a windowing event is handled with explicit locking when using pessimistic locks.

```
while (e = getEvent ()) {
  switch (e.type) {
    ...
    case MOUSE_DOWN:
      lock(x);
      ...       // modify x
      lock(y);
      ...       // modify y
      unlock(y) ;
      unlock(x) ;
      ...
  }
}
```

2. Changing the semantics of the locks can require a significant restructuring of the application code. The following example shows how the above case would be handled with optimistic locks.

```
    case MOUSE_DOWN:
      optLock(x );
      ...       // modify x
      optLock(y );
      ...       // modify y
      if (!assumptionValid(x)) {
      ...       // recovery code
      }
      if (!assumptionValid(y)) {
      ...       // recovery code
      }
      unlock(y) ;
      unlock(x) ;
```

3. The application developer has to be aware of the potential for deadlocks if locks on multiple objects are acquired.

In contrast, as computation is encapsulated as operations, and operations are very short-duration, an *ordering algorithm* can undo (by jumping back to an earlier checkpointed state or using an undo operation) and redo an operation as many times as needed. Undo and redo is important for optimistic algorithms, and to break deadlocks by aborting execution. This approach is also different from general optimistic schemes in distributed systems, like HOPE [8], which cannot make assumptions about short encapsulated operations, and therefore provide language-level constructs to express optimism and recovery.

4 Why not Replicated Databases ?

Looking at the data characteristics, it may seem that update operations are similar to transactions, and that a traditional replicated database should suffice for these applications. Indeed, some ideas can be borrowed from replicated databases, but there are many key differences which lead to different algorithmic design choices.

- *Short and incremental operations :* This results in the following

  1. *One notification per operation*: As operations are of short duration, the source site can send only one notification for that operation. This gives us an interesting and very important choice, which is not available in most replicated databases; that of sending the operation itself instead of its *effect*. By *effect* we mean that only the writes, i.e. the parts of objects with new values, are propagated. This is extremely important for optimistic protocols because it decouples undos at the source and other sites.
  2. *No explicit aborts*: The applications do not abort any operation. Any application level undo of an operation is done by issuing another operation.

- *Non-persistent Data:* The underlying algorithm can ensure that if one replica site receives an operation, and does not crash for a sufficient amount of time, every other replica site that is running will eventually receive that operation. This allows us to *locally commit* an operation. Local commit would be problematic if explicit aborts were permitted, or if the data being managed was persistent, which is not true in our case. Persistence is outside the scope of the algorithms considered in this paper. However, users can save their replica to a persistent store. And this can be used later for non collaborative work, or to initialize other o-sets. Unlike in conventional databases, we believe that leaving this to the application is acceptable.

- *Operation execution not interleaved at a replica:* This is unlike traditional (non-replicated) databases, which deal with concurrent operation execution, and allows simplification in algorithm design. More importantly, this implies that the operation access information is only used for deciding on an operation ordering that

results in replica convergence. Therefore, the access information provided to the algorithm can be factually incorrect, but enough to guarantee convergence. For example, consider rotation about the elbow joint for an articulated human model. Assume that the angles of the joints below the elbow, in the tree representation of the articulated model, are not affected by the rotation. Even though this rotation is changing the positions of these joints, it is commutative with other rotations about joints. Only a modification to the constraint on the elbow joint is not commutative with this rotation. Therefore, this update needs to only say that it accessed the elbow joint. However, if operations were allowed to interleave this access information would not be enough.

- *Predeclared or Execution-time access information*: Unlike databases which primarily use execution-time access information, many groupware applications can quite accurately predict the accesses of their updates (especially in light of the previous bullet). Some databases do use predeclared information as a supplement to execution-time information as it allows them to relax 2-phase locking by releasing locks early. Predeclared access information allows us to totally decouple ordering and execution, which can significantly reduce aborts and speedup commit in certain algorithms.

Database researchers have looked at the requirements of *asynchronous groupware* [2, 1], and proposed some solutions, and so it is instructive to understand the differences between their approach and the one we argue for. Their research has primarily looked at how to allow cooperation between long-lived transactions. This requires relaxing the atomicity requirement of transactions. Synchronous groupware is completely opposite, with very short-lived atomic operations. This can be partly explained by the observation that sharing between participants in asynchronous groupware is rarer and at certain well defined points. For example, checking in the updated source code of a code module is not frequent. These well-defined points usually correspond to commit of a long-lived transaction. The need for cooperation between these transactions arises because sometimes a transaction has not committed at a sharing point, but another transaction needs to read something written by this uncommitted transaction. For example, checking in a certain code module needed by another user, which corresponds to releasing a lock early in a long-lived transaction. Also, cooperation between transactions is usually triggered by the users. Another way of looking at the difference, is by considering the cooperation between users as contention between transactions/operations. It seems that contention in asynchronous groupware is rarer, but could last for a longer duration. Therefore, user intervention is feasible. In contrast, contention in synchronous/interactive groupware may be more frequent and transient. Therefore, the algorithms have to adapt to these situations with minimum user intervention.

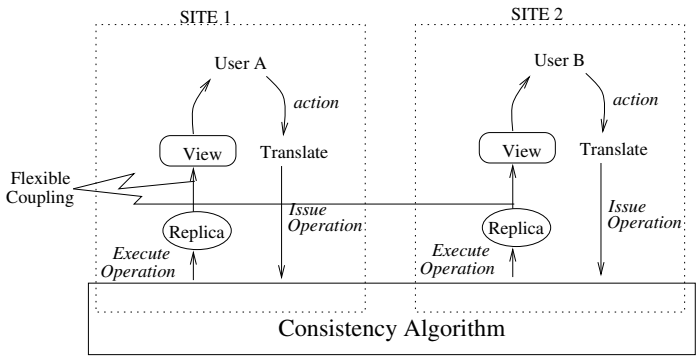


Figure 4: System Architecture

5 General Algorithmic Structure

Figure 4 shows a simplified view of the system architecture. The user performs an action which is translated into an operation by the application[4] and then issued to the consistency algorithm, which schedules it for execution. Read-only operations are only executed on the local replica, while others are executed at all replicas. The notification from the replica to the view depends on their coupling and is discussed in section 5.4.

5.1 Global Operation Ordering

Most algorithms define a global ordering on the operations that modify the shared data, by timestamping them at the issuing site. After timestamping, an operation is distributed to all sites including the issuer, which then execute it. This is a simplified structure which is useful for describing the high-level characteristics of algorithms. However, we will see many variations later, for example, in some cases timestamping and execution at the source overlap, in other situations the timestamping may abort etc. We also assume that the operation itself, and not its effect, is distributed to all sites. We discuss this assumption in some detail in section 5.3.

The global ordering defined by the timestamps can be a total order, but this is usually unnecessary. Therefore we assume that the global ordering is a partial order i.e. the ordering relation is irreflexive and transitive (a total order is a special case of a partial order). If a site executes operations in a total order that *respects* this global order, it is *correct*. The global ordering should satisfy certain properties, for example :

1. *Convergence Property*: Execution of the operations in any total order that respects the global order will result in the same state. This property is necessary for the replicas to converge.

2. *Process Ordering Property*: Operations issued by a site/process are globally ordered in the order they were issued.

[4]We omit this translation step from future discussion, and only say that the user issued an operation.

3. *Causal Ordering Property*: The global ordering satisfies causality.

All the algorithms discussed in section 6 satisfy the first two properties.

An operation e *commits* at a site p, when p knows that it has executed all operations preceding e in the global order. If a site executes operations only after they commit, it is guaranteed to be *correct*. This is where it is important to distinguish between *optimistic* and *pessimistic* algorithms. A pessimistic algorithm only executes an operation at p after it commits at p, therefore it is guaranteed to be correct. An optimistic algorithm does not wait for commit, however it might have to *reorder* operation execution, by undoing and redoing operations, when it notices that it has made a mistake. If this reordering causes surprise to the user, it is said to have caused *jitter* in the view. Of course, in some cases, surprise, and hence jitter, is subjective. However, the main reason why optimistic algorithms are not always suitable is the potential for *jitter*, which may not be acceptable to the user in certain situations. A secondary reason is that if a lot of reordering is necessary, it may be computationally too expensive. Note that this reordering is totally local to a site.

Another design choice for optimistic algorithms is to only detect potential divergence of replicas, and to provide mechanisms to the users to manually fix it.

5.2 Nature of Timestamping

Orthogonal to the optimistic versus pessimistic choice is the nature of the timestamps assigned to the operations.

1. *Access Independent, Dependent timestamps*: Independent timestamps do not depend on the data an operation accesses or the nature of the access i.e. read or write. For example, a global total ordering of all the operations can be defined by using Lamport clocks [13] and site identifiers to timestamp operations. Independent timestamps by default define a total order. Dependent timestamps use the access information of an operation for timestamping and therefore can define a partial order which is not total. A weaker ordering *can* lead to faster commit if the overhead to come up with this ordering is not too high.

2. *Timestamp Precision* : This property is useful to determine when an operation has committed. Informally, a timestamp is *precise* when by knowing the timestamps of the already received operations, and by looking at the timestamp of a newly received operation, say e, a site can accurately determine whether there are any operations preceding e in the global order that have not yet been received. The advantage of precise timestamps is that a site needs to only receive messages from sites which have issued operations preceding e in the global order. Therefore, a temporarily disconnected site cannot slow down commit of other sites. Usually, an imprecise timestamp implies a need to communicate with everyone for commit. Timestamps based on Lamport clocks are imprecise, because clocks at different sites may generate the same timestamp for concurrently issued operations. However, if a central server with a counter is used to assign timestamps to

all operations, by incrementing the counter by one after each timestamp assignment, the resulting timestamps are independent and precise. Vector clocks used for timestamping, which define a causal order on the operations, are precise (However, just causal ordering does not always satisfy the convergence property).

3. *Instantaneous timestamping* : An algorithm does instantaneous timestamping if it never needs to communicate with other sites to assign a timestamp to an operation. Precision and instantaneousness are at cross purposes. Optimistic protocols benefit from instantaneous timestamping as operations can be quickly timestamped and distributed.

4. *Aborts* : It is possible for the timestamp assignment to be aborted in certain algorithms (An example is the algorithm used in the DECAF system, discussed in Section 6). Timestamping of an operation e can be aborted when the timestamping is only partially completed, or after it is finished. In the first case only the issuer is affected as the operation has not yet been distributed to other sites, and usually occurs for dependent and precise timestamps. In the second case others are affected too, as the operation has been distributed to other sites. Timestamp abortion can increase commit time, and when using execution-time access information causes execution undo at the issuer which can result in jitter.

Ideally, we want an algorithm that uses dependent, precise timestamps with timestamping that is always instantaneous and abort-free. Unfortunately, precision and instantaneousness are conflicting requirements. However, not all timestamping properties are equally important in all interaction scenarios, therefore a suite of algorithms which satisfy different scenarios have been developed.

Before discussing the actual algorithms in Section 6, we discuss some other issues relevant to all ordering algorithms.

5.3 Distributing effect or update

In the above discussion we assumed that the update operation itself was disseminated. This means that the logic of the update, i.e. the function relating the object values read to the object values written is disseminated. Another option would be to only disseminate the *effect*, i.e. the object values written by an update. There are some situations in which propagation of the effect may be necessary.

1. *Partial Replication*: Assume that an o-set is somehow only partially replicated, i.e. not every replica site has all the objects that are members of the o-set. Although partial replication is not considered in this paper, it can be important in certain applications. Suppose a site issues an update which reads objects a and b and then computes $a = a + b$ and $b = a * b$. Of course, this site has replicas of both a and b. However, if this update is directly propagated to another site which doesn't have one of the two objects, it cannot be executed there, as the object to be read is not available. In such cases, disseminating the effect becomes essential.

2. *Heterogeneity*: In certain heterogeneous scenarios it may not be possible to execute an update at other sites, because of different machine architectures, operating systems etc. However, this is becoming less of a problem with the wide commercial availability of common execution environments like the Java Virtual Machine.

However, disseminating the effect has certain disadvantages.

1. *Coupling Rollback*: In an optimistic protocol, every site, including the source, executes the update optimistically. The effect of the optimistic execution at the source is what is disseminated to other sites and applied there. Therefore, if the source has to undo the execution of this operation, all the sites have to undo too. This coupled rollback also increases network communication as the effect has to be sent again.

2. *Delaying Commit*: In a pessimistic protocol, the source cannot send out the effect until it executes the update. And it cannot execute the update until it commits at the source. This cyclicity, though not a disaster, can cause slowdown in commit. Consider the following pathological case of a protocol using Lamport clocks. Three operations $e1, e2, e3$ are issued concurrently by sites 1, 2 and 3, each of whose current Lamport clock value is 50. Therefore $e1, e2, e3$ are assigned timestamps $(1, 50), (2, 50), (3, 50)$ respectively, with $(1, 50) < (2, 50) < (3, 50)$. The first operation to commit will be $e1$. Only after $e1$ commits and its effect is received by site 2, can $e2$ commit and its effect be sent out. $e3$ will commit only after both the effects of $e1$ and $e2$ have been received. The sequential commit behavior in this example greatly slows down commit.

5.4 Replica-View Coupling

In reality, users do not see the state of their replica directly, but through views displayed on some output device (like a computer screen), and it is the consistency and responsiveness of this view, and not the data, that is relevant to the user. Therefore, the nature of the *coupling* between the replica and the views is very important. In standard single user applications, whenever there is a change in the state of certain data, the view(s) attached to that data are notified, which then display the new state. If such an *immediate coupling* is used in the presence of replication, the characteristics of the data consistency algorithm are directly exposed to the user. Weakening the coupling can be useful for hiding or transforming some characteristics of these algorithms. For example, an optimistic consistency algorithm may need to reorder some operations by undoing and redoing their execution. It is useful to hide the intermediate stages in this undo and redo from the user. Weakened coupling should continue to guarantee that on quiescence, the view shows the current state of the replica. Therefore, with consistency algorithms in which the replicas converge at quiescence, the views will also converge.

The DECAF system [20] pioneered the concept of flexible coupling and is probably the most extreme example of hiding the characteristics of the underlying data consistency algorithm. DECAF uses an optimistic consistency

algorithm with undo and redo to guarantee convergence of data. The programmer specifies the shared data in a declarative manner by composing it from the DECAF supplied primitive objects, using DECAF's composition constructs. Relying on this total knowledge of the data being shared, the algorithm efficiently maintains multiple versions of the data. This allows complete control of what data versions to show in the view. DECAF supports two types of views: optimistic and pessimistic. Pessimistic views only show the effect of committed transactions, while optimistic views show the current state of the replica.

5.5 Initializing a new Participant

A new participant, i.e. one who is connecting to the set, needs to get an initial state of the set and should receive all operations that are ordered after this initial state. It is easy to get the initial state from any one of the current participants, but the critical problem is ensuring that the new participant does not miss any operations i.e. operations ordered after the initial state. This may require some synchronization between sites. We briefly outline two possible solutions.

In a system with a central server for routing all messages between participants, the solution to this problem is relatively simple. This central server can maintain the state of the set, either by actually executing the operations on its own replica, or by periodically requesting the replica state from a participant. In the latter situation, it can log all the operations that have been sent by the participants but not executed on the replica state it has. A new participant connects to the set by sending a message to this central server. The server replies with the replica state and the log of operations. In addition, all operations received by it after sending this reply are also forwarded to this new participant.

In a system with direct peer-to-peer communication, a flush protocol can be used to synchronize the participants, before letting the new participant join. During the flush, no new operations may be issued, so that the new participant does not miss any operations. A flush protocol, like the one used for virtual synchrony in ISIS [6], can also be a building block for providing fault tolerance. In situations where some of the current participants are only loosely connected with the rest, the flush may take a while to complete. In such cases, the requirement that no operations be issued during the time the flush is in progress may be too severe on the participants. Therefore, it may be necessary to utilize weaker synchronization protocols, like weak virtual synchrony [10].

6 Classification of proposed ordering algorithms

This section classifies and discusses the replica management algorithms described in the interactive groupware research literature. Figure 5 shows the classification according to the nature of timestamping.

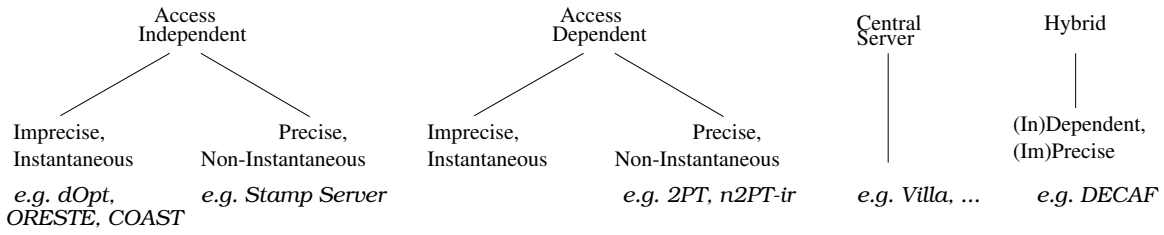


Figure 5: Algorithm Classification

6.1 Central Server

This is the simplest approach, and has been used in innumerable systems. It uses a central server, to which all operations are sent by the clients. The server distributes the operations received to all the clients (including the client which issued that operation). This imposes a very simple total ordering on the operations. Despite its simplicity of implementation and simple bounds on RT and UUT, it suffers from drawbacks as described in section 1.1. We consider one example of such a system, Villa, because of its unique consistency (or lack thereof) approach.

Villa The Villa system [5] provides an abstraction very similar to the o-set abstraction described in this paper. However, it only allows predeclaration of access information. An o-set can be either optimistic or pessimistic. For both, the server is used to construct the global ordering of the operations. In pessimistic mode, an operation is executed only after it is received from the server. In optimistic mode, an operation issued at a site (say p) is immediately executed and also sent to the server. A log of the access information of each operation executed by p is kept at p. When an operation that it issued is received from the server, p examines this log to see if there have been any conflicts which could have caused replica divergence. If yes, it notifies the application about which objects may no longer be consistent. The Villa system provides a cloning and reinitialization mechanism for the application/user to fix the incorrect replicas.

6.2 Independent, Precise

We consider a classic algorithm, again one which has been implemented in innumerable systems.

Stamp Server The basic algorithm is a variant of the central server algorithm. A server maintains a counter used to timestamp all operations. A client sends a request to the server for a timestamp, which returns the next value of the counter. This is used by the client to timestamp the operation, and then send it to other clients. Assuming that one-way latencies between clients and between clients and the server is l, the minimum possible RT and UUT is $2l$ and $3l$ respectively. However if the number of active participants is very low, it can be improved by dynamically

moving the counter to active sites.

6.3 Independent, Imprecise

dOpt Ellis and Gibbs [9] were one of the first to use replication to improve response time in groupware. They consider a programming model in which the shared data, for example a document, is represented by a single object replicated at each participant's site. Operations issued by participants are used to modify this shared data. They choose not to use explicit locking for maintaining consistency for various reasons, one of them being that choosing the granularity of the locks may be a problem. They guarantee causal ordering between operations using vector clocks. However, this causal ordering is not sufficient to guarantee convergence, as causally concurrent operations which modify the same data may be executed in a different order at different replicas. To solve this problem they use an operation transformation algorithm (dOpt), in which an operation may be transformed before execution such that executing two operations in different orders at replicas does not cause divergence. They provide transformation operations for a simple text outline editor, Grove. Later, researchers independently discovered an error in the algorithm. Cormack [7] has formalized the notion of operation transformations and provides a correct algorithm. An advantage of this approach is that undoing operations is never necessary. However, operation transformation for convergence also has certain disadvantages. We discuss this topic in more detail in section 7.1.

ORESTE Karsenty and Beaudouin-Lafon [12] describe the ORESTE algorithm, which considers the shared data to be a set of objects which is fully replicated at all the participant sites. Operations can add objects to the set, delete objects from the set, or can update a single object in the set. They define a total ordering of the operations using timestamps based on Lamport clocks. However, each site executes operations optimistically, therefore it might receive operations with a lower timestamp after it has executed a certain operation. When this happens, undo and redo of operations may be needed to guarantee replica convergence. They use information about commutativity and masking of operations to minimize undos. Note that the timestamping is *instantaneous* and *abort-free*. The optimistic approach provides instantaneous response with the potential for jitter. Also, as ordering is based on Lamport clocks, communication with every site is required for commit.

COAST The COAST system [18] fully replicates all the objects which are part of a shared document. It removes the restriction of ORESTE by supporting transactions with ACID properties, which can access a subset of the objects. The ordering of these transactions is done using an algorithm similar to ORESTE. However, it is not clear from the paper if or how commutativity and masking information for these transactions is provided by the application.

6.4 Dependent, Precise

2PT Adapted from a classic database approach, the performance of this algorithm is analyzed with synthetic workloads in [4]. It supports the o-set abstraction, and allows both predeclared and execution-time access information. Both read and write accesses can be specified, but for simplicity of discussion we consider only writes. The algorithm uses a token per object. A token has a version number, and the version numbers form a continuous sequence of integers starting with 0. These version numbers are used to timestamp updates which modify that object. For example, suppose the current version number of token A is 3, and token B is 1. Then if user Alice issues an update (say u) which modifies both object A and B, then after acquiring both tokens, this update will be timestamped with ((A,4),(B,2)). After timestamping, the tokens are released, and this update is multicast to others. Update u will be executed at a site only when update(s) with timestamps (A,3) and (B,1) have been executed. Note, that tokens are only useful for timestamping, and are not held until an update commits. Therefore, despite their similarity with locks, they are called tokens. To ensure no contradictory ordering information, tokens that are acquired to timestamp an update are not released until the update is fully timestamped. This is similar to the 2-phase locking scheme used in databases, and therefore the algorithm is said to do 2-phase timestamping or 2PT.

n2PT-ir This algorithm is a variation on the previous one, and relaxes the 2PT requirement for updates by releasing the tokens immediately. However, it only works for execution-time access information. For example, again consider update u and another concurrent update u' which also modifies A and B, and assume that token A is acquired before token B. Suppose u acquires the token for A before u' and gets timestamp (A,4). Now, if u immediately releases the token A, before acquiring token B, u' will get timestamp (A,5). As execution-time information is being used, the execution of u' will now block, until its site has executed the update with timestamp (A,4) i.e. u. This implies that u' does not get to that stage of its execution where it accesses B, until u is executed at its site (Note, that this would not be true if the execution of each update caused multiple notifications, where each notification corresponds to the new value of an object, to be sent out). Therefore, there is no danger that u' can acquire token B before u, which would cause a cycle in the ordering relation.

6.5 Hybrid

DECAF The DECAF system [20] is the first to recognize the need to consider both data and view consistency, and provides algorithms that manage replicated objects and their views. It provides a declarative model in which shared objects can be composed from the primitive shared objects like integer, real and string, provided by their framework. Shared objects are updated using transactions (only execution-time information is used), with the restriction that the issuer have replicas of all the objects accessed by the transaction. However, unlike the previous algorithms, other sites may not have all the objects accessed by a transaction. Therefore, the algorithm distributes

the effect and not the actual transaction to the other sites.

The underlying data consistency protocol is optimistic. The issuing site, say p, executes the transaction and assigns it a timestamp using a Lamport clock. It then sends the effect and this timestamp to all other sites. There is a primary site for each object which validates the timestamp for a transaction which reads or writes that object. The issuing site waits for validation responses from the primary sites of all the objects the transaction read or wrote. This validation gives some dependent properties to the basically independent timestamp. When all the primary sites say `yes`, the timestamp is approved, and this `ok` is sent by p to all sites which had been sent the transactions effect. If even one primary site says `no`, the transaction is aborted at p and at all other sites. Then p, has to retry by executing and timestamping again. In the case that a timestamp is approved, i.e. a site has received an `ok` message from p, the site goes and checks with all the primary sites to ensure that it has not missed an operation which had been ordered earlier. This avoids the need to wait for commit using the usual Lamport clock rules. However, it adds a round-trip communication with every primary copy by every site that received an `ok` message. Even if p is the primary copy for all the objects accessed by its transaction, the lower bound on UUT for a pessimistic view is $2l$. If p is not the primary copy for even one of the objects accessed by it, the lower bound on RT for a pessimistic view is $2l$. Another drawback of this approach is that there is no upper bound on the number of times a transaction may abort. Concurrent timestamping of two conflicting transactions can also cause both to abort. Each transaction requires a minimum of two multicasts, one to distribute the timestamped operation, and the second to inform about abort or commit. Overall, the behavior of the algorithm is quite complex.

7 Miscellaneous Issues

7.1 Operation Transformation

Operation transformation was pioneered by Ellis and Gibbs in Grove [9], and is used for optimistic execution. Cormack formalized it in [7], and it has been used in numerous text editors [21] and a spreadsheet [15].

This approach deals with two problems. Firstly, because of concurrent issue of updates, some operations may need to be transformed to preserve the intent of the user. For example, changing the position of insertion or deletion in a text buffer when there are concurrent inserts or deletes which are ordered earlier. This corresponds to the *after* function in Cormack's calculus, and is a technique which is useful for any situation in which operations are issued concurrently (even single copy systems). Secondly, it is used to provide convergence of replicas in optimistic execution without the need for undo and redo. This corresponds to the *before* function in Cormack's calculus.

The primary advantage of providing replica convergence through operation transformation is the potential for faster overall execution at a site, by substituting transformation for operation undo and execution. Suppose an operation u is received at a site where n operations that were issued concurrently with u have already been

executed. Cormack's algorithm requires a total of $2n$ transformations, of which n are done on u, and 1 each on the n operations that have already executed. Then the transformed u will be executed. Now consider an undo/redo approach, where u was supposed to be executed before the last $n/2$ of these operations (a valid average case assumption). Firstly, the last $n/2$ are undone. Then, u is transformed with respect to the first $n/2$ using the *after* function. Then, the last $n/2$ operations will be transformed wrt to u using the after function. Finally, u and the last $n/2$ operations will be executed. Therefore, the undo/redo approach causes n transformations, $n/2$ undos, and $n/2 + 1$ operation executions. In contrast, the operations transformation approach causes $2n$ transformations and 1 operation execution.

However, transformations as a replica convergence approach have some disadvantages. Firstly, *before* functions can be hard to come up with, and have to be proved correct. Secondly, before functions cannot implement many semantics which undo/redo can because of loss of information. For example, a before function cannot compensate for an operation that deletes some data (like deleting some characters in a text editor), because the data deleted has been lost. In contrast, with undo/redo, the operation can store what it deleted and can restore that when undone.

7.2 Access Granularity

Multigranularity locking has been used in database systems [3], and also in centralized groupware systems [14]. Its usefulness lies in preventing a mismatch in lock granularity and access granularity. This can be generalized to multigranular access information in the context of the o-set abstraction. However, the benefits for replicated data are not immediately apparent. Consider a 2PT algorithm which uses this multigranular access information to acquire tokens. Having many fine-grained locks can reduce the locality of acquiring tokens by a site. For example, in the first sequence compatibility table given in [14], which can be used for text editors, insertion of every consecutive character causes a new token to be acquired. This is acceptable for a centralized system, but may not be appropriate if acquiring a token requires communication with other sites. Also, in scenarios with data-dependent access patterns, the granularity of what will be accessed by an operation is not known beforehand, and so too many tokens may be acquired.

Despite this, multigranular access is a promising approach, and how to adapt it to ordering protocols and its performance in a distributed environment needs more investigation.

8 Conclusion

The algorithm requirements for interactive groupware are varied, with the application scenario, and network latency being the primary factors affecting the choice of algorithm to use. Despite the many types of replica management algorithms proposed, there is an unstated common theme in them. This paper illustrates the common data sharing requirements of interactive groupware and comes up with a data abstraction which is appropriate for these

requirements. We discuss why conventional replicated databases are not appropriate, and describe a general algorithmic structure for implementing this abstraction. The important design choices and characteristics of algorithm instances are described. This is used to roughly classify the algorithms described in the groupware research literature. The goal of this paper is that an understanding of the fundamental design choices and their implications, will stimulate more research in such algorithms. At the same time, a common terminology for describing algorithm characteristics will help in algorithm classification.

References

[1] D. Agrawal, J. L. Bruno, A. E. Abbadi, and V. Krishnaswamy. Relative serializability: An approach for relaxing the atomicity of transactions. In *Proceedings of the 13th ACM Symposium on Principles of Database Systems*, 1994.

[2] N. S. Barghouti and G. E. Kaiser. Concurrency control in advanced database applications. *ACM Computing Surveys*, 23(3):269–317, 1991.

[3] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[4] S. Bhola, G. Banavar, and M. Ahamad. Responsiveness and consistency tradeoffs in interactive groupware. In *Proceedings of 7th ACM Conference on Computer Supported Cooperative Work*, November 1998. To appear.

[5] S. Bhola, B. Mukherjee, S. Doddapaneni, and M. Ahamad. Flexible batching and consistency mechanisms for building interactive groupware applications. In *Proceedings of the 18th ICDCS*, 1998.

[6] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[7] G. V. Cormack. A calculus for concurrent update. Technical Report CS-95-06, Department of Computer Science, University of Waterloo, Ontario, 1995.

[8] C. Cowan and H. L. Lutfiyya. A wait-free algorithm for optimistic programming: Hope realized. In *Proceedings of the 16th International Conference on Distributed Computing Systems(ICDCS)*, pages 484–493, 1996.

[9] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In *Proceedings of the ACM SIGMOD'89*, pages 399–407, 1989.

[10] R. Friedman and R. van Renesse. Strong and weak virtual synchrony in horus. Technical Report TR95-1537, Computer Science Department, Cornell University, 1995.

[11] J. P. Granieri, J. Crabtree, and N. I. Badler. Production and playback of human figure motion for visual simulation. *ACM Transactions on Modeling and Computer Simulation*, 5(3):222–241, July 1995.

[12] A. Karsenty and M. Beaudouin-Lafon. An algorithm for distributed groupware applications. In *Proceedings of the 13th ICDCS*, pages 195–202, 1993.

[13] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[14] J. Munson and P. Dewan. A concurrency control framework for collaborative systems. In *Proceedings of the 6th ACM CSCW*, 1996.

[15] C. R. Palmer and G. V. Cormack. Operation transforms for a distributed shared spreadsheet. In *Proceedings of 7th ACM Conference on Computer Supported Cooperative Work*, November 1998. To appear.

[16] A. Prakash and H. S. Shim. Distview: Support for building efficient collaborative applications using replicated objects. In *Proceedings of the 5th CSCW*, 1994.

[17] M. Roseman and S. Greenberg. Building real time groupware with groupkit, a groupware toolkit. *ACM Transactions on Computer Human Interaction*, 1996.

[18] C. Schuckmann, L. Kirchner, J. Schummer, and J. M. Haake. Designing object-oriented synchronous groupware with COAST. In *ACM CSCW'96*, 1996.

[19] S. Singhal. *Effective Remote Modeling in Large-Scale Distributed Simulation and Visualization Environments*. PhD thesis, Dept. of Computer Science, Stanford University, August 1996.

[20] R. Strom, G. Banavar, K. Miller, A. Prakash, and M. Ward. Concurrency control and view notification algorithms for collaborative replicated objects. *IEEE Transactions on Computers*, 47(4):458 – 471, April 1998.

[21] C. Sun and C. Ellis. Operational transformation in real-time group editors: Issues, algorithms and achievements. In *Proceedings of 7th ACM Conference on Computer Supported Cooperative Work*, November 1998. To appear.